State diagrams

- Finite State Machines
  - State diagrams
    - States
    - Events
    - Transitions
  - Parsing input with state diagrams

State Machines

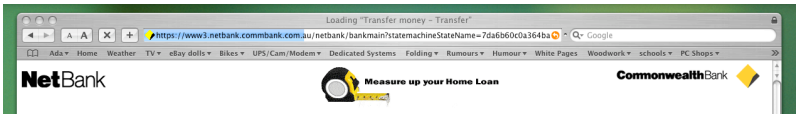
- Interaction diagrams allow you to model the interactions between components
- IDs don't specify what happens inside a component.
- A Finite State Machine
  - specifies the internal behaviour of a class.
  - models the lifetime of a single object.
  - lists the (finite set of) states an object exists in
  - lists the events an object receives
  - shows how an object responds to those events

Examples of states

- An insurance policy can exist in various states - quotation, creation, existing, claim processing, fraud investigation, terminated.
- A heating system can be in various states as well...
  - inactive, heating-up, active, cooling-down.
- In both cases we can draw a map of the states and the valid transitions between states.
- We can also specify the circumstances in which an object changes from one state to another, and what actions (if any) are taken when that happens.

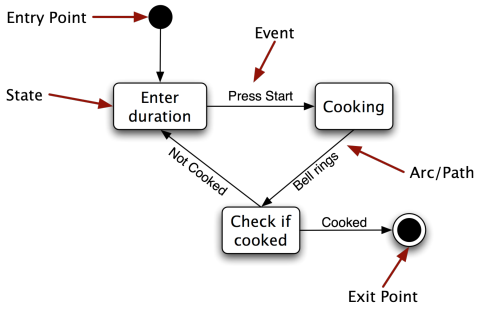
State Machines

- A state machine consists of
  - States
    - One start state, and possibly a final state.
  - Directed arcs (or paths) between the states
  - Events
- They can also contain
  - Conditions on the events
  - Actions that are executed on state transition



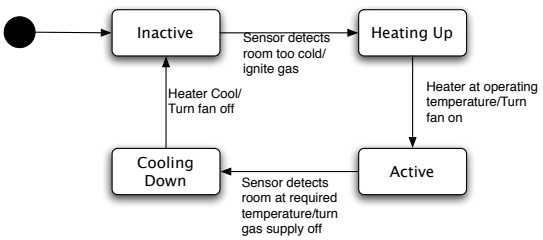
State diagram overview

- An example state diagram for a person using a microwave oven.



Heater state diagram

- The heater state diagram not only shows the events that cause a state transition, but also the action taken when that transition occurs (e.g. "ignite gas", "Turn fan on").



States

- A state is a condition or situation during which an object satisfies some condition, performs some activity or waits for some event.
- The normal view of a state diagram has
  - The object doing nothing when in a state, and only performing some action when an event occurs.
  - It then moves to the next state where it again does nothing but wait for the next event.
- Good way to think about states when designing them.
- State diagrams now allow for some actions while waiting in a state.

States

- A state has several parts. All but the name are optional.
  - Name
  - Entry/Exit actions
  - Activities (actions that occur while in a state)
  - Internal transitions (changes that don't cause a state transition)
  - Substates (not covered)
  - Deferred events (not covered)

Events

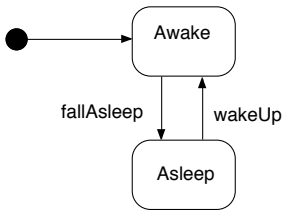
- Something that happens at a point in time.
- An event has no duration
- Concurrent events are two events that are related only by coincidence, (eg 2 planes take off at the same time) and have no effect on each other.
- An event is a one-way transmission of information from one object to another. An object sending an event to another object may expect a reply, but the reply is a separate event under control of the second object, which may or may not choose to send it.
- Every event is a unique occurrence, but they can be grouped into event classes with attributes if necessary (eg aeroplane flight departures)
- Often events are simple signals, (eg key depressed), but events often also have attributes.

Transitions

- A transition has five parts...
  - Source state
  - Event trigger
  - Guard condition (the event only "fires" if this is true)
  - Action
  - Target State
- The format for a transition is
  - event [guard]/action

Simple state machine

- States are a period of time when the system stays (much) the same.
- Events occur instantaneously, as do state transitions.



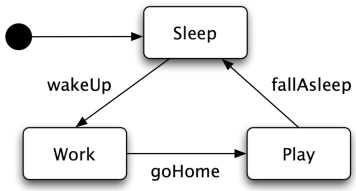
Implementing a state diagram

- An implementation of a state diagram is a state machine.
- A method call can be an event.
- It may cause a state transition.

```
public class Person {

   private final static int ASLEEP  = 0;
   private final static int AWAKE = 1;

   private int state = AWAKE;

   public void wakeUp () {
      state = AWAKE;
   }

   public void fallAsleep () {
      state = ASLEEP;

   }
}
```

A 3 state machine

- Events are only responded to when we are in an appropriate state.
- Any implementation must ensure that spurious events are ignored.
- E.g. Should not respond to fallAsleep events when at work.



A 3 state machine

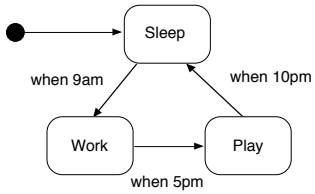
```
public class Person {
   private final static int SLEEP  = 0;
   private final static int WORK   = 1;
   private final static int PLAY   = 2;

   private int state = SLEEP;

   public void wakeUp () {
      if (state == SLEEP) { // ensure
         state = WORK;
         System.out.println ("I'm working");
      }
   }
   public void fallAsleep () {
      if (state == PLAY) {
         state = SLEEP;
         System.out.println ("I'm asleep");
      }
   }
   public void goHome () {
      if (state == WORK) {
         state = PLAY;
         System.out.println ("I'm playing");
      }
   }
}
```


Indirect event triggers

- Events may not always be triggered directly by a method call.
- We may use the value of an attribute (or a combination of them) as the encoding of a state.
  - An object can go into a many states - a 32 bit integer can have over 4 billion of them!
  - We subset the states into those we are interested in.
  - A method call may cause a change in attributes that are then recognised as a suitable change of state.

Indirect event triggers

- Consider a class that has a "tick" method called every hour.
- It maintains an internal hour attribute
- Only when the hour changes to specific values do we want to change state.
  - The "play" state is associated with an hour value of 17-22 (5pm - 10pm)



Indirect event triggers

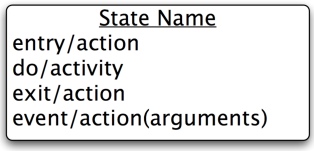
```
public class Person {

   // Shows a state machine in which the states are triggered as
   // a by product of external events.
   // The class has to recognise what events could trigger a state
   // change, and examine the appropriate attributes to determine
   // if a state change is appropriate.

   public  final static int SLEEP = 0;
   public  final static int WORK  = 1;
   public  final static int PLAY  = 2;
   private final static int MAX_STATE = PLAY;
   private final static int START_STATE = SLEEP;


   private String stateNames[] = {"Sleep", "Work", "Play"};

   private int state = START_STATE;
   private int hour = 0; // 24 hour clock
```


Indirect event triggers

```
   public void tick () {
      // each call to tick causes the clock to go forward by an hour
      hour++;
      if (hour >= 24) {
         hour = 0;
      }

      switch (hour) {
         case 9:
            state = WORK;
            break;
         case 17:
            state = PLAY;
            break;
         case 20:
            state = SLEEP;
            break;
         default:
            // do nothing
            ;
      }
   }
```

States with internal actions

- A state is generally considered to be a period of time during which an object is inactive.
- Sometimes we want to model some continuing action within a state, as well as actions that occur as we enter or leave the state.
- UML provides notations the following notation to describe these.



States with internal actions

- A security system for a house will be in a number of states, such as off, self test, and on.
- When in the on state it is constantly monitoring any sensors.
- It issues a special beep when it is activated, and a different beep when deactivated.

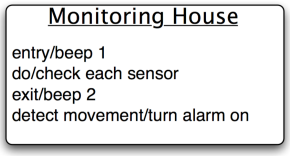
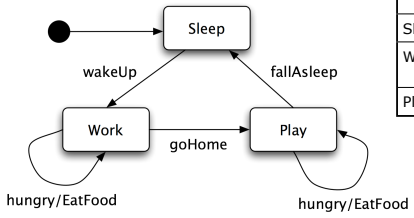


Table representation of a state diagram

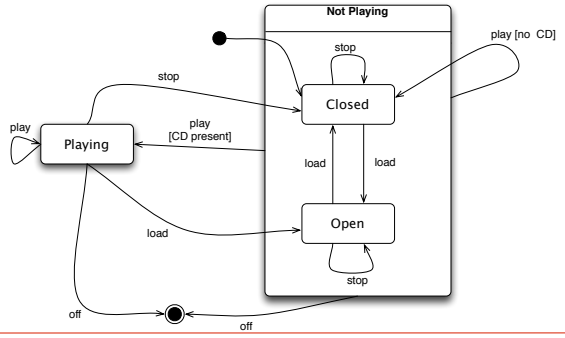
- A state diagram can be thought of as a large table, indexed by state and event.



	wakeUp	goHome	fallAsleep	hungry
Sleep	Work	-	-	-
Work	-	Play	-	Work/Eat
Play	-	-	Sleep	Play/Eat

Composite states

- Sometimes multiple states respond to events in the same way - they all lead to one other state.
- Instead of replicating all of the transitions we group the states into a composite state.

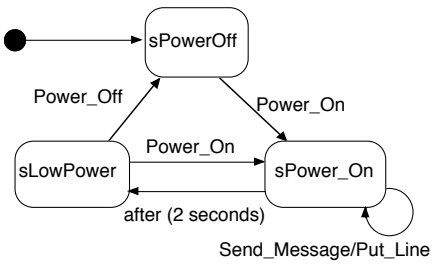


Time and Change events

- A time event is activated upon the passing of time
  - "after 2 seconds"
- A change event occurs upon a boolean condition becoming true
  - when condition
    - when item full
    - when time = 10pm

Time events

- You can use time based conditions to trigger state changes.



Parsing words

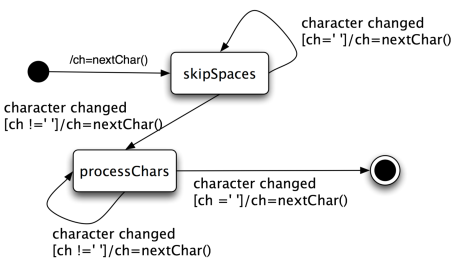
- A common use of state machines is in parsing text files - breaking up the input into substrings.
  - a Java compiler
  - StringTokenizer class.
- The following example shows a simple state machine for parsing sequences of non blank characters from a String.
- The program is not particularly robust - it is quite easy to extend the state machine to fix the problems.

Parsing input

- When parsing input we use a single character which helps encode the state.
  - The values the character can take is split into different states
    - character, digits, punctuation, spaces etc.
- An event is when this character changes.
- The character typically changes on each transition.

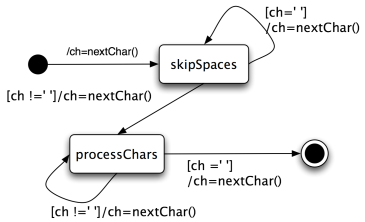
Parsing input

- This state machine skips past spaces and displays each non blank character until it encounters another space or the end of input.
- As you can see all state transitions have the same event (character changed) and differ only in the guards



Parsing input

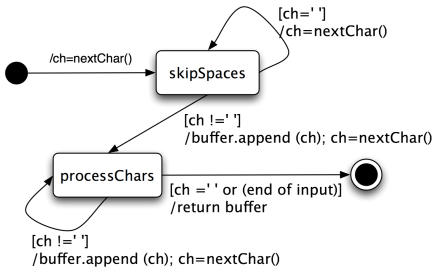
- In this case it is easier to ignore the (common) event, and just consider states and guards.



	ch=' '	ch != ' '
skipSpaces	skipSpaces	processChars
processChars	exit	processChars

Parsing words

- To get the characters that are parsed, you need to store them in a buffer as the state machine works.



Parsing words

```
public class Tokenizer {

   private String words;
   private int upto = 0;

   public Tokenizer (String words) {
      this.words = words;
   }
   public String nextToken () {

      final int inSpaces = 0;
      final int inWord = 1;
      final int collectedWord = 2;

      final String stateNames [] = {"inSpaces", "inWord", "collectedWord"};

      final int START_STATE = inSpaces;
      final int END_STATE = collectedWord;

      StringBuffer word = new StringBuffer ();
```

Parsing words

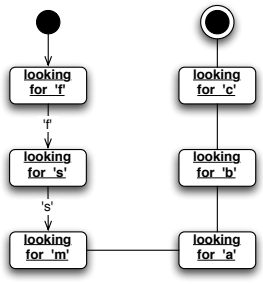
```
int state = START_STATE;
while (state != END_STATE) {
   switch (state) {
      case inSpaces:
         if (words.charAt (upto) == ' ') {
            upto++;
         } else {
            word.append (words.charAt (upto));
            upto++;
            state = inWord;
         }
         break;
```


Parsing input

```
            case inWord:
               if (words.charAt (upto) == ' ') {
                  state = END_STATE;
               } else {
                  word.append (words.charAt (upto));
                  upto++;
               }
               break;

            default:
               ;
         }
      }
      return word.toString ();
   }
}
```


Regular Expression Pattern Matching

- Regular expressions such as "fsm*abc" are converted into state machines when they are executed.
- "fsm*abc" should match against any of the following...
  - fsmabc
  - fsmxxabc
  - fsmababc
  - fsmababababc
- The state diagram for this...



Summary

- Finite state machines are a very useful tool in some applications.
- They help you to remember to account for all possible transitions into and out of a state.
- They are not always a suitable way to document the behaviour of an object.
- As always you need to determine if you will get value from using finite state machines before spending a lot of time on them.

Using states in software

- States are a good way to model the internal behaviour of software.
- People are not always good at using interfaces that have states unless it is very apparent what state they are in.
  - Aircraft accidents when pilots have failed to realise the plane is in autopilot
  - The Unix text editor vi has two states - insert mode and command mode (confusion as to what mode you are in causes endless problems!)